Trees

Algorithms and data structures ID1021

Johan Montelius

Fall term 2023

Introduction

Linked lists might be useful but the true value of linked data structures comes when we have more complex structures; the next step up from a linked list is a tree structure. It's called a tree since the structure originates in a *root* that then divides into branches that are further divided into *branches*. A branch that does not divide further is terminated by a *leaf*.

The trees that we will work on now are so called *binary trees* i.e. a branch always divides into two branches if it does not end in a leaf. The operations that we will look at are: construction, adding, searching for and removing an item. We will later look at more general tree structures but the principles are the same.

A binary tree

Let's construct a binary tree where each node in the tree has: a key, a value, a left branch and a right branch. The keys could be anything but to be able to talk about sorted trees we require them to be *comparable*. The keys are mapped to values and we assume a functional mapping i.e. any given key is mapped to exactly one value. As an example we could have used kth-id as keys and names as values but in this example we will simply use integers as keys and integers as values.

We use an nested class `Node` to describe the internal structure of the tree. The tree it self only holds one property, the `root` of the tree. If the root is a null pointer the tree is empty. A node with both left and right branches set to `null` is a leaf in this representation; it makes life easier.

```
public class BinaryTree {

    public class Node {
        public Integer key;
```

```
        public Integer value;
        public Node left, right;

        public Node(Integer key, Integer value) {
            this.key = key;
            this.value = value;
            this.left = this.right = null;
        }
    }

    Node root;

    public BinaryTree() {
        root = null;
    }
    :
    :
```

Now assume that that the tree is sorted so all nodes with keys smaller than the root key are found in the left branch and the nodes with larger keys in the right branch. The ordering is of course recursive so if we go down the left branch we will find smaller keys to the left etc. Note that we sort based on the key, the value could be anything.

Now implement two methods:

- `add(Integer key, Integer value)` : adds a new node (leaf) to the tree that maps the `key` to the `value`. If the key is already present we update the value of the node. Note that the tree should still be sorted.
- `lookup(Integer key)` : find and return the value associate to the `key`. If the key is not found we return `null`.

When implementing the add-method one could chose to implement it recursively. The algorithm would look like follows, start in the root node:

- Is the key of the node equal to the key that we add - change the value and return.
- Is the key that we add less than the key of the node and
  - we have left branch - recursively add the key value to the left branch and return,
  - if not - create a new node and set it as the left branch and return.
- Same thing for right branch.

The lookup-method becomes very similar in its structure i.e. recursive traversal of the tree in order to find the value that we are looking for. Set up a benchmark and compare the execution time for a growing data set. Note that when you construct a binary tree you should not construct it using an ordered sequence of keys - what would happen if you did? How does the lookup algorithm compares to the binary search algorithm that you used in one of the previous assignments?

Depth first traversal

Very often you want to go through all items that you have in a tree and the question then arise in what order you should traverse the tree. If the tree like in our example is ordered with smaller keys to the left, one natural order would be to traverse the item starting with the leftmost and then work your way towards the rightmost. This order is referred to as *depth first* i.e. you go down as deep as possible to the left before considering the alternatives.

A simple example of this can be shown by adding a print method to the node class and print the keys and values in a depth first order.

```
public void print() {
    if(left != null)
        left.print();
    System.out.println(" key: " + key + "\tvalue: " + value);
    if(right != null)
        right.print();
}
```

When we implement the print method we make use of the implicit stack of the java language. When we recursively have printed the items of the left branch we return to the node we are currently in, print the key and value and then recursively print the items of the right branch. Can we do this using an explicit stack? You don't have to implement it but think about what it would look like.

Assume that we want to traverse the items in a tree in a depth first, left to right, order but we want to get the items one by one instead of all items in one go. We want to create a data structure that can hold the tree and also a position where the next item can be found if requested. This data structure is often referred to as an *iterator* and these are so common that a programming language often have special constructs to handle them.

An iterator

In Java we can define a class that implements a `Iterator`. An iterator has two methods: `hasNext()` and `next()`, and this is sufficient to loop through

the values. There is a bit of boiler plate code to implement an Iterator but this should get you started.

The first thing we do is to declare `BinaryTree` to implement the `Iterable` interface. The class should then provide a method that returns an `Iterator`. We will in this example return an iterator that produces a sequence of integers i.e. the values of the nodes in the tree.

```
public class BinaryTree implements Iterable<Integer> {
  :
  :

  public Iterator<Integer> iterator() {
    return new TreeIterator();
  }
  :
}
```

The question is of course what the `TreeIterator` class should look like. The boring solution would be to traverse the whole tree and store all values in an array but this is of course very inefficient if we have a large tree and in the end only want to extract the first few values. A more fun implementation is to keep the tree as it is and then keep track of where we are in the tree.

The `TreeIterator` class will have two attributes: the *next* node and a *stack* to keep track of where to go next. You need an implementation of a stack and why not use the one you implemented in a previous assignment. In the code below we assume that we have a generic implementation that can be specified to be a stack of nodes.

In the end you will have to implement the constructor and the three methods of the iterator interface that we override. We will not handle removing values so we throw an exception in case this method is used.

```
  public class TreeIterator implements Iterator<Integer> {

    private Node next;
    private Stack<Node> stack;

    public TreeIterator() {
      :
    }

    @Override
    public boolean hasNext() {
      :
    }
```

```
    @Override
    public Integer next() {
      :
    }

    @Override
    public void remove() {
      throw new UnsupportedOperationException();
    }
  }
```

Now how do we implement the iterator?

Using a stack

Assume that the `next` attribute is always pointing to the next node, the value of which we shall return if the `next()` method is used. If the attribute is a null pointer the iterator has reached the end of the sequence.

The first value that we should return is of course the leftmost value in the tree. This means that we should initialize the data structure by walking down the left branch as far as possible.

Assume now that the leftmost node is a leaf (both left and right branch are null), then how can we move to the parent node when we have delivered the next value in the sequence? This is where the stack comes into play, if we push nodes on the stack as we go down the left branch we can pop the stack and find the parent node.

The only tricky part is when the `next` node has a right branch that is not null (the left branch is either null or a branch that we have already traversed). In this case you need to walk down to the leftmost node in the right branch. As you go you of course push nodes on the stack since these nodes are nodes that you should return to later.

Note - the node on the stack will not necessarily be the parent node of the `next` node. It will be a node on the path towards the root whose value has not been returned and whose right branch is still to be explored. When you return a value of a node and move down its right branch the node itself will not be pushed on the stack.

If everything works you should now be able to use the tree in a *for-each* loop. Since the `BinaryTree` implements the `Iterable` interface this should work:

```
        BinaryTree tree = new BinaryTree();

        tree.add(5,105);
```

```
        tree.add(2,102);
        tree.add(7,107);
        tree.add(1,101);
        tree.add(8,108);
        tree.add(6,106);
        tree.add(3,103);

        for (int i : tree)
            System.out.println("next value " + i);
    }
```

In you report explain in your own words how you implement the methods of the `TreeIterator`. Also describe what will happen if you create an iterator, retrieve a few elements and then add new elements to the tree. Will it work, what is the state of the iterator, will we loose values?